Extracted from:

Programming Erlang, Second Edition

Software for a Concurrent World

The Pragmatic Bookshelf

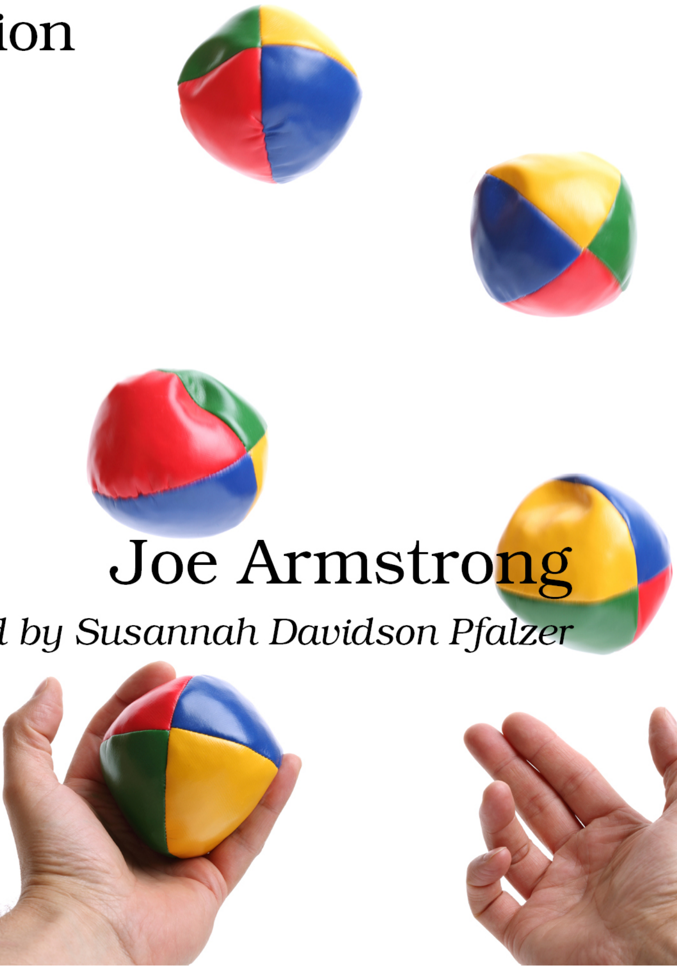
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Second Edition



Joe Armstrong

Edited by Susannah Davidson Pfalzer

Programming Erlang, Second Edition

Software for a Concurrent World

Joe Armstrong

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-53-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2013

CHAPTER 18

Browsing with Websockets and Erlang

In this chapter, we will see how to build applications in the browser and extend the idea of using message passing to outside Erlang. This way, we can easily build distributed applications and integrate them with a web browser. Erlang thinks that the web browser is just another Erlang process, which simplifies our programming model, putting everything into the same conceptual framework.

We're going to pretend that a web browser is an Erlang process. If we want the browser to do something, we'll send it a message; if something happens within the browser that we need to attend to, the browser will send us a message. All of this is possible thanks to *websockets*. Websockets are part of the HTML5 standard and are bidirectional asynchronous sockets that can be used to pass messages between a browser and an external program. In our case, the external program is the Erlang runtime system.

To interface the Erlang runtime system to websockets, we run a simple Erlang web server, called *cowboy*, to manage the socket and the websocket protocol. Details of how to install cowboy are covered in Chapter 25, *Third-Party Programs*, on page ?. To simplify things, we assume that all messages between Erlang and the browser are JSON messages.

On the Erlang side of the application these messages appear as Erlang maps (see Section 5.3, *Maps: Associative Key-Value Stores*, on page ?), and in the browser these messages appear as JavaScript objects.

In the rest of this chapter, we'll look at six example programs, including the code that runs in the browser and the code that runs in the server. Finally, we'll look at the client-server protocol and see how messages from Erlang to the browser are processed.

To run these examples, we need three things: some code that runs in the browser, some code that runs in an Erlang server, and an Erlang server that understands the websockets protocol. We're not going to look at all the code here; we'll look at the code that runs in the browser and in the server but not the code for the server itself. All the examples can be found at https://github.com/joearms/ezwebframe. The browser code in the examples has been tested only in the Chrome browser.

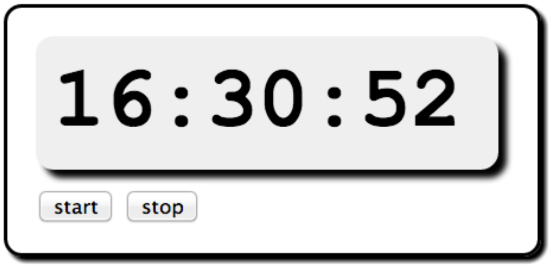
Note: The code shown here is a simplified version of the code in the ezwebframe repository. The code here is written using maps. The code in the repository is kept in sync with the Erlang distribution and will reflect any changes to Erlang when maps are introduced in version R17 of Erlang (expected in late 2013, but maps will appear in branches on GitHub before the official release).

To run the code yourself, you'll need to download the code and follow the installation interactions. As far as we are concerned, the interesting parts of the code are the part that runs in the browser and the part that runs in the server.

All the examples use a simple technique for controlling the browser from Erlang. If Erlang wants the browser to do something, it just sends the browser a message telling it what to do. If the user wants to do something, they click a button or some other control in the browser and a message is sent to Erlang. The first example shows in detail how this works.

18.1 Creating a Digital Clock

The following image shows the clock running in a browser. All the irrelevant details of the browser window, such as the menus, toolbars, and scrollbars, are not shown so that we can concentrate on the code.



The essential part of this application is the display. This contains a time, which is updated every second. From the Erlang point of view, the entire browser is a process; so, to update the clock to the value shown earlier, Erlang sent the browser the following message:

```
Browser ! #{ cmd => fill_div, id => clock, txt => <<"16:30:52">> }
```

Inside the browser, we have loaded an HTML page with a small fragment of HTML like this:

```
<div id='clock'>
  ...
</div>
```

When the browser receives a fill_div, it converts this into the JavaScript command fill_div({cmd:'fill_div', id:'clock', txt:'16:30:52'}), which then fills the content of the div with the required string.

Note how the Erlang message containing a frame gets converted to an equivalent JavaScript function call, which is evaluated in the browser. Extending the system is extremely easy. All you have to do is write a small JavaScript function corresponding to the Erlang message that you need to process.

To complete the picture, we need to add the code that starts and stops the clock. Putting everything together, the HTML code looks like this:

websockets/clock1.html

```
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./clock1.css" type="text/css">
<body>
  <div id="clock"></div>
  <button id="start" class="live_button">start</button>
  <button id="stop"  class="live_button">stop</button>
</body>
<script>
$(document).ready(function(){
   connect("localhost", 2233, "clock1");
   });
</script>
```

First, we load two JavaScript libraries and a style sheet. clock1.css is used to style the display of the clock.

Second, there is some HTML that creates the display. Finally, we have a small fragment of JavaScript that is run when the page is loaded.

Note: In all our examples we assume some familiarity with jQuery. jQuery (http://jquery.com) is an extremely popular JavaScript library that simplifies manipulating objects in the browser.

websock.js has all the code necessary for opening a websocket and connecting the objects in the browser DOM to Erlang. It does the following:

1. Adds click handlers to all buttons with class live_button in the page. The click handlers send messages to Erlang when the buttons are clicked.
2. Tries to start a websocket connection to http://localhost:2233. On the server side, the function clock1:start(Browser) will be called in a freshly spawned process. All this is achieved by calling the JavaScript function connect("localhost", 2233, "clock1"). The number 2233 has no particular significance; any unused port number over 1023 would do.

Now here's the Erlang code:

websockets/clock1.erl
```
-module(clock1).
-export([start/1, current_time/0]).

start(Browser) ->
    Browser ! #{ cmd => fill_div, id => clock, txt => current_time() },
    running(Browser).

running(Browser) ->
    receive
        {Browser, #{ clicked => <<"stop">>} } ->
            idle(Browser)
    after 1000 ->
            Browser ! #{ cmd => fill_div, id => clock, txt => current_time() },
            running(Browser)
    end.

idle(Browser) ->
    receive
        {Browser, #{clicked => <<"start">>} } ->
            running(Browser)
    end.

current_time() ->
    {Hour,Min,Sec} = time(),
    list_to_binary(io_lib:format("~2.2.0w:~2.2.0w:~2.2.0w",
                                 [Hour,Min,Sec])).
```

The Erlang code begins execution in start(Browser); Browser is a process representing the browser. This is the first interesting line of code:

```
Browser ! #{ cmd => fill_div, id => clock, txt => current_time() }
```

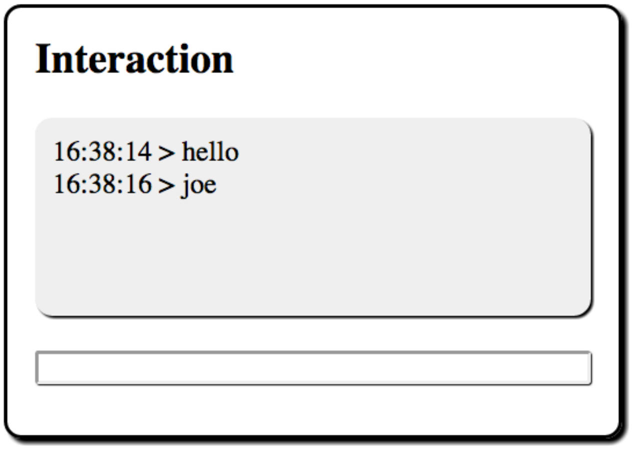
This updates the display. I've repeated this line for emphasis. My editor told me to remove it. But no. To me this is very beautiful code. To get the browser to do something, we send it a message. *Just like Erlang.* We've tamed the browser. It looks like an Erlang processes. Whoopeee.

After the initializing, clock1 calls running/1. If a {clicked => <<"stop">>} message is received, then we call idle(Browser). Otherwise, after a timeout of one second, we send a command to the browser telling it to update the clock and call ourselves.

idle/1 waits for a start message and then calls running/1.

18.2 Basic Interaction

Our next example has a scrollable text area for displaying data and an entry. When you enter text in the entry and press the carriage return, a message is sent to the browser. The browser responds with a message that updates the display.


Interaction

16:38:14 > hello
16:38:16 > joe

The HTML code for this is as follows:

websockets/interact1.html
```
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./interact1.css" type="text/css">
<body>
  <h2>Interaction</h2>
  <div id="scroll"></div>
  <br>
  <input id="input" class="live_input"></input>
</body>
<script>
$(document).ready(function(){
   connect("localhost", 2233, "interact1");
   });
</script>
```

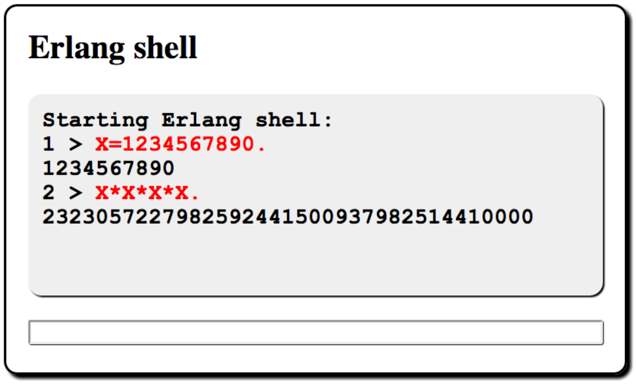
And here is the Erlang:

websockets/interact1.erl
```
-module(interact1).
-export([start/1]).

start(Browser) -> running(Browser).

running(Browser) ->
    receive
        {Browser, #{entry => <<"input">>, txt => Bin} }
            Time = clock1:current_time(),
            Browser ! #{cmd => append_div, id => scroll,
                        txt => list_to_binary([Time, " > ", Bin, "<br>"])}
    end,
    running(Browser).
```

This works in a similar manner to the clock example. The entry sends a message containing the text in the entry to the browser each time the user hits Enter in the entry. The Erlang process that manages the window receives this message and sends a message back to the browser that causes the display to update.

18.3 An Erlang Shell in the Browser

We can use the code in the interface pattern to make an Erlang shell that runs in the browser.


Erlang shell

```
Starting Erlang shell:
1 > X=1234567890.
1234567890
2 > X*X*X*X.
2323057227982592441500937982514410000
```

We won't show all the code since it is similar to that in the interaction example. These are the relevant parts of the code:

websockets/shell1.erl
```
start(Browser) ->
    Browser ! #{cmd => append_div, id => scroll,
                txt => <<"Starting Erlang shell:<br>">>},
    B0 = erl_eval:new_bindings(),
    running(Browser, B0, 1).
running(Browser, B0, N) ->
```

```
    receive
        {Browser, #{entry => <<"input">>}, txt => Bin}} ->
            {Value, B1} = string2value(binary_to_list(Bin), B0),
            BV = bf("~w > <font color='red'>~s</font><br>~p<br>",
                    [N, Bin, Value]),
            Browser ! #{cmd => append_div, id => scroll, txt => BV},
            running(Browser, B1, N+1)
    end.
```

The tricky bit is done in the code that parses and evaluates the input string.

websockets/shell1.erl

```
string2value(Str, Bindings0) ->
    case erl_scan:string(Str, 0) of
      {ok, Tokens, _} ->
        case erl_parse:parse_exprs(Tokens) of
          {ok, Exprs} ->
            {value, Val, Bindings1} = erl_eval:exprs(Exprs, Bindings0),
              {Val, Bindings1};
            Other ->
              io:format("cannot parse:~p Reason=~p~n",[Tokens,Other]),
                        {parse_error, Bindings0}
        end;
      Other ->
        io:format("cannot tokenise:~p Reason=~p~n",[Str,Other])
    end.
```

And now we have an Erlang shell running in the browser. Admittedly it's a very basic shell, but it illustrates all the techniques necessary to build a much more sophisticated shell.